2

IDA PAPER P-2701

SOFTWARE TESTING INITIATIVE FOR STRATEGIC DEFENSE SYSTEMS

Bill Brykczynski, *Task Leader*

Reginald N. Meeson
Christine Youngblut
David A. Wheeler

March 1992

Prepared for
Strategic Defense Initiative Organization

IDA
INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 92-041141

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 1992	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Software Testing Initiative for Strategic Defense Systems		5. FUNDING NUMBERS MDA 903 89 C 0003 Task T-R2-597.21
6. AUTHOR(S) Bill Brykczynski, Reginald N. Meeson, Christine Youngblut, David A. Wheeler		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard St. Alexandria, VA 22311-1772		8. PERFORMING ORGANIZATION REPORT NUMBER IDA Paper P-2701
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) SDIO Room 1E149, The Pentagon Washington, D.C. 20301-7100		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution: January 11, 1993.		12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

SDIO is planning to develop and deploy a ballistic missile defense system called Global Protection Against Limited Strikes (GPALS). Testing of GPALS software will play a critical role in ensuring the cost-effective development and reliable execution of the GPALS system. An earlier IDA study examined the states of the art and practice in the field of software testing. The study identified deficiencies in SDI software testing capabilities, and recommended that SDIO launch a technical initiative to address these deficiencies. The current study lays the groundwork for this initiative. Emphasis is placed on three areas: (1) improving current and near-term SDI software testing practices by promoting consistent use of effective technology, (2) identifying promising new testing techniques that can add significantly to the reliability, cost-effectiveness, and quality of SDI software, and (3) fostering research into several fundamental problems of testing large-scale, concurrent, distributed, real-time, and fault-tolerant software systems.

14. SUBJECT TERMS Software Testing, Technology Transition, Global Protection Against Limited Strikes (GPALS), SDI, Requests for Proposal (RFP).			15. NUMBER OF PAGES 78
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

IDA PAPER P-2701

SOFTWARE TESTING INITIATIVE FOR STRATEGIC DEFENSE SYSTEMS

Bill Brykczynski, *Task Leader*

Reginald N. Meeson
Christine Youngblut
David A. Wheeler

March 1992

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.21

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) for the Strategic Defense Initiative Organization (SDIO), under contract MDA 903 89 C 0003, Subtask Order T-R2-597.21, "Software Testing of Strategic Defense Systems." The objective of the subtask is to assist the SDIO in planning and monitoring software testing research, development, and practices.

In support of this objective, IDA had previously developed a high-level concept of a research and development initiative that would address deficiencies in required SDI software testing technology. The purpose of this paper is to provide SDIO with a set of recommendations and supporting analysis to begin a comprehensive software testing initiative in fiscal year 1993.

This paper was reviewed by the following members of IDA: Dr. Robert J. Atwell, Dr. Dennis W. Fife, Dr. Randy L. Garrett, Dr. Karen D. Gordon, Ms. Audrey A. Hook, Dr. Richard J. Ivanetich, and Mr. Terry Mayfield.

EXECUTIVE SUMMARY

With the Missile Defense Act of 1991, Congress appropriated funds for the development and deployment of ballistic missile defense systems capable of protecting the United States, as well as US forces, friends, and allies overseas. Several of the experimental weapon and sensor programs funded by the Strategic Defense Initiative Organization (SDIO) will make the transition to full-scale development in the next few years. All of these systems will rely upon software to function properly.

Historically, 50% to 80% of the cost of software development is spent in testing. There is no reason to expect that testing will play a lesser role in Strategic Defense Initiative (SDI) software development. Clearly, software testing will play a critical role in developing reliable and cost-effective SDI software. However, previous studies by the Institute for Defense Analyses (IDA) have suggested that conventional testing methods sufficient for small-scale sequential software may not be adequate for testing software for strategic defense systems. These methods are largely ad hoc and may not scale up to the levels required by SDI systems. In addition, there are currently no concerted efforts to ensure that existing advanced methods for testing software are used within the SDI program.

This paper lays the groundwork for a technical initiative that will develop and deploy software testing technology needed to ensure the development of reliable and cost-effective software for SDI. Three objectives for this initiative have been identified:

a. To ensure that all element program offices have consistent approaches to testing SDI software.

b. To experiment with new and improved testing techniques and to facilitate their transition into SDI standard practice.

c. To strengthen the scientific basis for software testing methods, tools, and metrics related to the SDI software testing domain.

A specific approach has been developed for achieving each of these objectives. These approaches involve technology transition projects, software testing experiments, and applied research projects.

Transition Projects

The first approach involves moving improved testing technology into common SDI practice. SDI systems will be developed by many different contractors and multiple Service agents. It is important that SDIO ensure all software is developed using consistent, effective testing techniques. The following four recommendations are intended to improve current SDI software testing practice.

SDIO should ensure that appropriate testing clauses are placed in all element Requests for Proposals (RFPs) and contracts.

The Department of Defense (DoD) software development standard, DoD-STD-2167A, does not provide sufficient specifications for testing SDI software. Although the standard describes a series of test-related documents that should be produced, it does not suggest or mandate methods for early detection and prevention of defects during the software development process. SDIO should develop a common set of RFP and contract clauses that describe mandatory contractor defect detection and prevention methods. This paper provides a sample set of RFP and contract clauses, as well as an explanation of, and rationale for, the clauses.

One of the most effective methods for detecting defects during software development is called *formal inspections*. Formal inspections involve a small group of participants who follow a set of procedures for reviewing a work product (e.g., a set of requirements, designs, or code). The purpose of the review is to identify as many defects as possible. Inspections often find up to 80% of code defects during development, and, when properly applied, can reduce overall system development costs. The use of formal inspections can also play an important role in achieving high software reliability. The contractor that developed the National Aeronautics and Space Administration (NASA) Space Shuttle software attributes much of that systems high software reliability to the use of formal inspections.

SDIO should advocate the use of formal inspections for all Global Protection Against Limited Strikes (GPALS) software. At this time, however, element program offices are not currently planning to use this advanced defect detection method. Developing RFP and contract clauses that describe the requirements for formal inspection will assist element program offices in establishing this method for GPALS.

SDIO should query element program offices to determine the potential for the use of defect prevention methods.

Defect prevention methods may represent a key driver in reducing software life-cycle costs and improving system reliability. Implementation of defect prevention programs is a task best suited to element program offices and their contractors. However, SDIO should provide element program offices and their contractors with information on the potential costs and benefits of defect prevention programs.

Error cause analysis is a promising defect prevention method that should be considered for use by SDIO. Error cause analysis requires examination of defects to identify their root causes. Preventive measures are then developed to reduce the likelihood of recurrence for that type of defect. SDIO should also review existing commercial educational programs for error cause analysis and, if appropriate, suggest that element program offices offer this training to their contractors.

SDIO should conduct an element program office workshop to promote the application of advanced testing tools and methods.

SDIO is evolving from a research program to a deployment program. With this evolution, a need exists for regular communication among element program managers and SDIO on the activities of software testing. However, element program offices are currently placing little emphasis on the processes and technology for testing SDI element software. A workshop would provide a useful forum to discuss element program office test plans, RFP and contract clauses, and so forth.

SDIO should establish and actively participate in a software testing subcommittee within the Computer Resources Working Group (CRWG).

The CRWG provides a regular forum for discussing a wide variety of SDI computer and software-related topics. A software testing subcommittee should function as an advisory group for element software testing computer and software-related topics. A software testing subcommittee should function as an advisory group for element software testing activities. Members of this subcommittee should have direct responsibilities for element testing activities. Suggested activities for this group include the review of element software testing plans and results of testing activities, discussion of available software testing tools, and discussion on lessons learned from application of testing technologies.

Evaluation Experiments

The second initiative approach involves experimenting with promising new testing techniques that can add significantly to the trustworthiness, reliability, and cost-effectiveness of SDI software. Many techniques have been convincingly demonstrated in research environments but need improvement and refinement before they can be broadly applied. These techniques may offer significant benefits to the SDI software development community, and it is important that SDIO investigate and experiment with them. This paper suggests that SDIO establish a series of technology evaluation experiments that are intended to:

a. Identify from the collection of available software testing techniques those that most reliably and cost-effectively detect the critical defects.

b. Bridge the gap between research and routine practice, and accelerate the transition of useful testing techniques.

c. Absorb the costs and mitigate the risks of applying new, unproven testing techniques.

d. Expedite feedback to researchers and technology developers on necessary improvements and refinements.

This paper provides the following recommendations for establishing a software testing experiment program:

SDIO should annually budget several software tesing experiments to identify promising new testing techniques.

Historically, SDIO has not provided funds for software testing experimentation. In fiscal year 1992, funds were allocated for an initial National Test Facility experiment with ANNA, a tool that can aid in detecting Ada coding defects. Such efforts should be encouraged, and it is hoped that additional experimentation will occur. However, the scope of experimentation must be broadened and expanded.

SDIO should solicit specific testing experiment proposals from development organizations (e.g., Army Strategic Defense Command, Air Force Space Systems Division) that play key roles in SDI software development. The proposals need significant input from the contractors, so the solicitations need to filter down to them. SDIO should also coordinate with Service research and development agencies, such as the Naval Research Laboratory and Rome Laboratory, to identify promising avenues of experimentation.

SDIO should provide recognition for innovations in software testing that evolve from these experiments.

Achieving the necessary levels of assured software performance and reliability will require identifying and applying new solutions to software testing problems. One of the fastest ways to advance the current state of testing practice is to reward innovative application of new, but unproven, tools and techniques. *Software testing activities are* sometimes viewed as counter-productive by development teams. As software development deadlines approach, improved testing can mean that more corrections and rework are required. Program offices, however, must take a broader view and recognize that improved testing leads to improved products. Rewards for innovation in testing may include letters of commendation, presentations to the SDI community, and notice in internal publications.

SDIO should plan to incorporate positive experimental results in technology transition activities.

The goal of these experiments is to identify technology that can improve current software testing capabilities. As positive experiment results appear, SDIO should ensure that the results are communicated to the SDI software development community. Avenues for this communication include presentations at CRWG meetings, updates to established software testing standards, and revisions to RFP and contract Statement of Work (SOW) requirements for software testing practices.

Research Projects

The third initiative approach involves solving several fundamental problems of testing large-scale, concurrent, distributed, real-time, and fault-tolerant software systems. Current methods and techniques for testing such systems may not provide the level of assurance of correct operation, safety, and reliability necessary for SDI software. Without solutions to these problems, SDI software is likely to contain latent defects that could compromise the success of SDI's mission. This paper provides the following recommendations for establishing a software testing research program.

SDIO should establish and fund a long-term research program to improve testing capabilities for large-scale, concurrent, distributed, real-time, and fault-tolerant software systems.

A sustained, long-term research program would likely need on the order of $1 million per year for five years. This amount is likely to produce useful results within the available time frame. Given the current shortage of active testing research investigations, it was judged that much more than this amount could not be applied productively toward SDI's needs. To revitalize an active software testing research community it is important that this program is recognized as a sustained, multi-year effort. Stop-and-start funding will defeat program objectives. Also spreading the money thinly over too many intermediate funding agencies could dilute the program's impact. By taking a leadership position in establishing this program, SDIO can expect other research funding agencies to cooperate and perhaps even contribute additional resources of their own to extend the program.

Once the research program is initiated, SDIO should closely monitor the progress and effectiveness of research projects. Methods for advanced experimentation and evaluation of research results should be developed.

Annual reviews of ongoing research projects should be conducted. Renewed funding should be contingent on demonstrated progress and on the expectation of useful results. In addition, annual workshops should be held where both research ideas and practical problems of testing SDI software can be presented and discussed. This mix of theory and practice should provide two-way communication between researcher and practitioner, accelerating the transition of new research results into practice and feeding practical utility and relevance information back into the research process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Purpose

With the Missile Defense Act of 1991, Congress appropriated funds for the development and deployment of ballistic missile defense systems capable of protecting the United States, as well as US forces, friends, and allies overseas [MDA 1991]. Several of the experimental weapon and sensor programs funded by the Strategic Defense Initiative Organization (SDIO) will make the transition to full-scale development in the next few years. All of these systems will rely upon software to function properly.

Software testing will play a critical role in developing reliable and cost-effective Strategic Defense Initiative (SDI) software. However, previous studies by the Institute for Defense Analyses (IDA) have concluded that conventional testing methods sufficient for small-scale sequential software may not be adequate for testing software for strategic defense systems [Youngblut 1989, Brykczynski 1990]. Moreover, there is no concerted effort to ensure that advanced methods for testing software are used within the SDI program.

This paper lays the groundwork for a technical initiative that will develop and deploy software testing technology and practices needed to ensure reliable and safe operation of computer-controlled strategic defense systems. The paper provides SDIO with a set of recommendations and supporting analysis to begin a comprehensive software testing initiative in fiscal year 1993.

1.2 Organization

Section 2 of this paper describes key SDI software testing issues, such as the current state of software testing practice, problematic software characteristics, and the ability to meet system development budgets and schedules. A brief overview of the software testing initiative, including the overall objectives, beneficiaries, cost and schedule, is presented in Section 3. Sections 4, 5, and 6 provide descriptions of technology transition projects, technology evaluation experiments, and research projects that make up the initiative. Specific recommendations for implementing these projects and experiments are included within each of these sections. Appendix A provides Request for Proposal (RFP) and contract clauses suggested for SDI element program offices, as well as rationale for these clauses. These clauses provide a mechanism to encourage improved software testing practices.

1.3 Background

The concept of an SDIO software testing initiative emerged in the summer of 1990. During that time, several briefings were made to SDIO officials on the anticipated difficulty of testing Phase I Strategic Defense System (SDS) software. It was recognized by SDIO that an effort to develop improved testing capabilities was necessary, and IDA was tasked to develop a description of such an effort.

In late 1990, IDA published a paper for SDIO which concluded that conventional testing methods sufficient for small-scale sequential software would not be adequate for testing SDS software [Brykczynski 1990]. This paper also provided an initial, high-level description of an SDIO software testing initiative and identified three classes of projects that should form the initiative:

a. Technology transition projects

b. Technology evaluation experiments

c. Research projects

Preliminary estimates of cost, schedule, and methods for implementation were included in the paper.

In 1991, SDIO re-focused the SDI program to provide ballistic missile defense from accidental launches and limited strikes. The new SDI architecture, Global Protection Against Limited Strikes (GPALS), preserved many of the software testing challenges found in the Phase I SDS architecture. Based on the initial 1990 paper, IDA began to refine the description of the components of the initiative and to develop specific recommendations for implementing the effort. This paper presents the results of that effort.

1.3.1 GPALS Segments

SDIO is currently planning the development and deployment of the GPALS system. GPALS will provide protection from an attack of up to 200 ballistic missiles on the United States as well as on US forces, friends, and allies overseas. GPALS is composed of three interoperable segments:

a. Theater Missile Defense (TMD) will provide transportable systems for theater ballistic missile protection. TMD elements include transportable ground-based radars and interceptors, and space-based sensors for launch detection and missile tracking.

b. National Missile Defense (NMD) will provide protection to the US from accidental or limited ballistic missile attacks. NMD elements include fixed ground-based radars and interceptors, as well as space-based sensors.

c. Global Missile Defense (GMD) is focused on developing space-based interceptors to assist TMD and NMD. The primary system being examined is the Brilliant Pebbles (BP) interceptor that will provide boost and midcourse detection and interception.

At the present time, specific system architectures for each segment have not yet been developed. Previously proposed SDI elements such as Ground Based Radar (GBR), Ground Based Interceptor (GBI), and Brilliant Eyes (BE) will be included in the segment architectures. Formal GPALS acquisition schedules have yet to be determined. Following the guidance of the Missile Defense Act of 1991 [MDA 1991], SDIO is planning to develop and deploy an early NMD capability by 1996. Establishing segment architectures and developing acquisition schedules will be a high priority in fiscal year 1992.

1.3.2 Overview of the Software Testing Initiative

SDI systems will critically depend on reliable software. Experience demonstrates, however, that the current state of software testing technology and practice poses a significant risk in building a reliable and affordable SDI system. Therefore, the overall goal of the software testing initiative is to provide the technology to substantially enhance the reliability, safety, and affordability of software developed to operate and control SDI element systems.

This section of the paper provides a brief overview of the objectives, beneficiaries, and cost and schedule of the testing initiative.

1.3.2.1 Objectives

To achieve the goals stated above, three specific objectives have been identified for the testing initiative. The first is to ensure that all element program offices have consistent approaches to testing GPALS software. Testing practices vary widely across the software industry. This variability contributes directly to increased program risk. Standardization of approaches offers several benefits. First, it can reduce the existing variability in testing practices that increases program risk. More specifically, standardization can ensure consistent application of testing techniques that are known to be cost effective. It can provide a common basis for analyzing reliability data across system elements. It can also ensure use of a minimum set of product and process measures that support early identification of potential problems. These measures can themselves be used to identify isolated good practices whose use should be encouraged across the program, thus helping to refine the set of standardized practices. Finally, standardization can reduce testing costs by allowing wide usage of a common set of testing tools. Included within this aspect of the initiative is the building or acquiring experience with robust production quality tools, providing training in proven testing technology, and promoting software defect prevention efforts within the SDI program.

The second objective of the initiative is to experiment with new and improved testing techniques and to facilitate their transition into GPALS standard practice. The primary approach used to accomplish this objective will be to foster experimentation with promising or emerging testing technology that has not yet been demonstrated as cost effective on large systems. Experiments may involve construction of prototype tools and the application of new, unproven techniques and tools in actual SDI development projects.

The third objective of the initiative is to strengthen the scientific basis for software testing methods, tools, and metrics related to the SDI software testing domain. For example, a problem facing software reliability and safety assurance is that even though all known software defects may be corrected before deployment, this does not imply that the software will operate perfectly. It could simply reflect, for example, that the tests were not adequate to expose hidden defects. Measuring the effectiveness of software testing and the potential added value of additional testing is an open research area. In addition, critical parts of SDI software will exhibit characteristics such as large-scale, real-time, concurrent, and distributed processing, yet the technology currently available to test such software is rudimentary.

1.3.2.2 Beneficiaries

Although the recommendations in this paper are being developed for SDIO, the testing initiative is primarily aimed at benefiting element contractors and their program offices. It is the element contractors who must apply appropriate testing processes during the development of element software, and it is the element program offices who bear the responsibility for ensuring that element software is safe, reliable, and affordable.

The testing initiative will influence element program offices in near, medium, and long-range timeframes. Initiative products, such as RFP and contract clauses specifying advanced testing practices, may have an immediate impact on how element program offices contract for software. Other aspects of the initiative will seek to improve program office awareness of software testing difficulties and solutions.

As described in Section 1.3.1, GPALS is composed of three segments. At this time it seems likely that an operational NMD segment will be deployed in the 1996 timeframe. For this system, several ground- and space-based elements will be developed for integration into new and existing command and control systems. Although formal software sizing estimates for this NMD system have not yet been produced, it is likely that NMD will require development of 8-10 million lines of software. As described in Section 2, testing this software will be an extremely difficult task. Initiative experiments that evaluate near-term improvements in testing technology will likely have a significant impact on testing NMD segment software. Experiments that evaluate longer-term testing research results will further improve methods for testing GMD segment software.

Research focused on testing real-time, concurrent, distributed, and fault-tolerant software may take several years before useful results can be applied in the field. A number of SDI elements and system enhancements, however, are likely to be in development well into the next decade. Research results, therefore, will be available to improve the testing of GMD segment software.

1.3.2.3 Cost and Schedule

The initiative should be planned initially as a five-year, $20 million effort. The five-year period provides a reasonable amount of time to move existing technology into standard practice. In addition, high-payoff research efforts should be able to return experimental results within this period. The success of the evaluation experiments will determine if individual element program offices and contractors continue them beyond the five-year initiative. The $20 million cost reflects the likely amount of useful work that could be accomplished within the five-year period.

This paper lays out the basic objectives for the initiative; defines criteria for acceptable experimental evaluation projects; identifies several promising, high-priority projects that should be undertaken; and proposes an infrastructure for initiating projects. The overall cost and schedule estimates are meant as guidelines. Detailed estimates for project cost and duration will depend on which projects are selected, who performs them, and how they are implemented. Specific annual initiative budgets will have to be developed as a part of the SDIO fiscal year budget process.

2. KEY SDI SOFTWARE TESTING ISSUES

The ability to develop SDI software with sufficient confidence that it will perform correctly has been raised as a key technical issue numerous times in the history of the SDI [Fletcher 1983, Cohen 1985, Parnas 1985]. Testing is the principal approach available for assuring correct software operation and, therefore, testing will play a critical role in building confidence in SDI[1] software. Three aspects of software testing are of importance to the SDI program:

a. Assuring reliable and safe operation of SDI software.

b. Effectively testing software (e.g., real-time, distributed, concurrent) that imposes particular testing problems.

c. Achieving required levels of assurance within cost and schedule constraints.

These points are expanded upon below, following an account of the current state of software testing art and practice.

2.1 Current State of Software Testing

Software testing involves the application of tools, techniques, and methodologies to determine the presence of defects.[2] These defects exist not only in code, but also in requirements, specifications, designs, documentation, and other products of the software development process. To date, the majority of software testing research and development has focused on relatively simple, small-scale, sequential software. Techniques that are effective for small pieces of software, however, often do not scale up well for large systems. Ballistic missile defense systems require software that exhibits real-time performance, fault tolerance, and distributed operation—all of which severely complicate the testing process. Technology to adequately test such software has not kept pace with the demands for these capabilities and remains largely a research topic.

1. This paper will use the term *SDI* when referring to general architecture concepts associated with strategic defense systems. The term *GPALS* will be used to refer to the specific architecture being examined by SDIO.

2. A *defect* is an error in software code, design, or requirements that might cause faults. A *fault* is a manifestation of a defect. A *failure* is a serious fault that cannot be recovered from and prevents a system from achieving its mission.

To complicate matters further, testing technology that is available is seldom applied. This was found in an effort in 1981 to document Department of Defense (DoD) software testing practices [DeMillo 1987]. This study found a *fifteen-year* gap between the state of the art in software testing and the state of DoD practice. There is no evidence to indicate that this gap has closed in recent years.

Testing is typically an ad hoc, labor-intensive effort only poorly supported by systematic procedures and automated tools. As shown in Figure 2-1, data collected during the development of a number of early, large software systems (e.g., Semiautomated Ground Environment (SAGE), Naval Tactical Data System (NTDS), GEMINI, SATURN V, and IBM OS/360) reveal that software unit, integration, and system testing alone represent approximately one half of the software development effort [Boehm 1970, Alberts 1976]. In the National Aeronautics and Space Administration (NASA) Apollo system, 80% of the software development effort was spent on testing [Dunn 1984]. Although software development methods and practices have generally improved over time, there is no evidence that the proportion of effort required for testing has been significantly reduced.

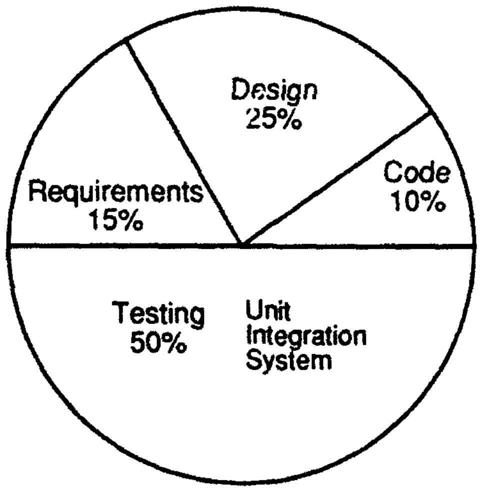

Figure 2-1. Proportional Costs of Software Development Activities

The classic questions that remain unanswered by current software testing technology are: "How much testing is needed?" and "Which techniques should be applied?" Testing budgets and schedules are typically based on experience gained in building similar systems. All too often, testing simply stops when the budget or schedule dictates.

2.2 Reliable and Safe Operation

Testing will be the primary means by which confidence in the reliable and safe operation of SDI software is gained during its development and probably throughout its deployed lifetime. Operational testing ordinarily represents only a small fraction of software testing. Although operational testing generally exercises the system's capabilities, it typically involves only a small portion of the software and reveals little about its reliability. This is because it is virtually impossible to control the operational testing environment to exercise the software more thoroughly. Exercising smaller software components in relative isolation enables much more rigorous testing. In addition, operational testing is very expensive. Defects discovered early in the development are much less expensive to correct [Boehm 1981, p. 40]. Therefore, software testing must be performed throughout the development process. Although latent software defects may still be exposed in operational tests, earlier testing efforts are generally much more effective and much less expensive.

Unfortunately, testing cannot guarantee the absence of *all* defects in software [Myers 1979, pp. 9-11; Boehm 1970, p. 25]. It is generally acknowledged that large, complex software systems will contain defects. The overall SDI design, therefore, must include consideration of the impact that software defects can have on reliable and safe system operation. The software system architecture and subsystem designs must also consider the potential impact of defects in lower-level software components. Testing can be used to guide the application of software fault-tolerance techniques to recover functionality and performance in the event of hardware and software faults, and to ensure continued safe operation.

Another problem arises from the difficulty of determining the reliability of software. Conventional measures of reliability, such as mean-time-to-failure, are based on statistical models originally developed for hardware systems. There are fundamental differences, however, between software and hardware failures. Software, for example, does not wear out or fail randomly. If software contains defects and fails to produce correct results, then given the exact same conditions, it will produce exactly the same incorrect results again and again. Current measures of software reliability actually reflect the probability of external conditions arising that cause software defects to produce incorrect results. They are not direct measures of defects and the probability of their producing incorrect results. That is, a system that has operated flawlessly for years can suddenly begin to fail when the distribution or timing of inputs change. Hence, the reliability of software that has not been modified in any way appears to change.

Statistical techniques have also been applied to estimate the number of defects remaining in software and the potential impact that software faults might have on overall system operation. For example, there are techniques for estimating the number of defects remaining in a program based on the history of defects already found. However, these techniques need considerable refinement to increase their accuracy.

Meanwhile, there is reason for hope. A number of testing techniques can conclusively demonstrate the absence of particular, limited types of software defects. For example, interface datatype mismatches have been eliminated by strong type checking in high-level programming languages such as Ada. These techniques provide for a high degree of confidence in certain aspects of the software and more such techniques are needed.

2.3 Problematic Software Characteristics

Conventional testing methods sufficient for simple, sequential software are not adequate for the SDI. Portions of the SDI software will have to be highly real time, concurrent, distributed, and fault tolerant. Each of these characteristics imposes particular testing problems, as noted in the following paragraphs.

a. **Large-scale software** consists of a million or more lines of source code. The complexity of such software systems, the diversity of possible inputs that must be handled, and the potential conditions under which faults might arise increase dramatically with the size of a system. The adequacy of testing, in terms of the number of possible input values and program states that can actually be checked, therefore, decreases significantly as system size increases. Current automated testing tools extend testing capabilities beyond what can be achieved manually, but are still not adequate for very large systems.

b. **Real-time software** must execute within strict time requirements. In real-time systems, a missed deadline is considered a fault even if computed values are correct. Consequently, testing must verify timing behavior as well as functionality. Time-critical computations commonly overlap and compete for processing resources. The timing of one function, therefore, depends on the resources required and the time consumed by all other functions that can interrupt its execution. Current analytical techniques, in practice, require making simplifying assumptions that can lead either to under-equipped systems that unexpectedly miss deadlines or to over-equipped systems that never fully use all the resources available. Testing is complicated by the time-critical events that have to be orchestrate to exercise the software under numerous possible conditions. In addition, "invasive" testing techniques that are commonly used to check functionality (e.g., executable assertions and probes that collect test coverage data during execution) often distort performance enough to introduce faults and to mask the effects of existing defects.

c. **Concurrent software** consists of multiple tasks or "threads" that execute in parallel on multiple processors or asynchronously on a single processor. Cooperation between tasks is managed by operations that allow two tasks to synchronize and exchange information. Defects in concurrent software include all those that can appear in single-thread sequential programs plus defects in synchronization. The difficulty of testing concurrent software arises from the nondeterminism inherent in synchronization events, which means that two executions of the same software with the same inputs may not produce the same results. All possible synchronization sequences must be analyzed and tested, which is further complicated by the problem of determining path feasibility. Special tools and techniques are required to repeat particular synchronization sequences to identify causes of faults and to check modifications made to remove defects.

d. **Distributed software** is made up of cooperating processes that execute on multiple processors separated physically and connected via communications channels. These processes can be modeled using the same techniques used for concurrent software. The main difference is that synchronization requires communication between processors, which typically introduces additional latency and uncertainty into the timing of synchronizations. This uncontrollable variability increases the difficulty of systematically repeating faulty behavior. External communication channels also represent additional points of potential failure that concurrent tasks on single processors and closely coupled multiprocessors typically do not need to address.

e. **Fault-tolerant software** is designed to ensure that system failures are controlled. Tolerance of hardware failures is usually achieved by the provision of redundant system components. Tolerance of software faults tends to focus on the use of multiple, independently-developed versions of the software (N-version programming) and fault detection and recovery mechanisms (e.g., recovery blocks). Testing such software requires introducing hardware and software faults in order to evaluate the response. Characterizing potential defects realistically, however, is difficult and may give rise to questions about the validity of such tests.

The only methods in current practice for testing these types of software are ad hoc and cannot be relied upon to scale up to the size and complexity anticipated for SDI software. The problems are well recognized, however, and the technology to address them is being investigated. For example, in the case of reproducible execution of concurrent software, basic support technology is under development that will provide a framework in which some existing testing techniques for sequential software can be applied.

2.4 Meeting Budgets and Schedules

Three avenues are available for increasing the effectiveness and productivity of software testing for SDI systems. A significant step forward can be achieved by emphasizing the use of testing tools for use in both software development and post-deployment support. There are a number of advanced prototype tools that are ready to be transformed into production-quality tools, or to be taken from a specialized development environment to one applicable for SDI systems. These tools should be integrated into the planned SDIO Software Engineering Environment (SEE). At the very least, a set of basic testing utilities such as test drivers, test data generators, coverage analyzers, test management tools, and regression testing tools should be used.

A second avenue for increasing the software testing effectiveness and productivity involves the cost-effectiveness of available testing techniques. The effectiveness of various techniques ranges from detecting all instances of particular, narrow but well-defined classes of defects (e.g., 100% of interface datatype defects), to detecting smaller percentages on wider ranges of defects (e.g., 60 to 90% of all types of defects via inspections). Although these defect detection rates seem promising, the techniques that yield these rates are not often used. The cost of implementing these techniques is often cited as the reason they are not used. A thorough understanding of the cost effectiveness of available techniques, including areas where techniques overlap (i.e., which techniques find the same types of defects), is needed to adequately plan and control testing activities for SDI software.

Theoretical approaches to evaluating the effectiveness of particular testing techniques are being pursued, but this research may not yield practical results in the next five years. In the meantime, empirical data collected from trial use of techniques on near-term SDI software development efforts could go a long way to filling this gap. Such data could provide cost-effectiveness information for a number of techniques that can then be used in selecting techniques to apply.

Finally, the use of a limited set of product and process measures could provide element contractors with better insight into testing activities. This insight could reveal potential schedule or cost slippages while there is still time to take appropriate action. Defect detection rate is one fundamental measure. Here the basic mapping of the number of defects found per thousand lines of code (KLOC) against time provides an indirect indication of both the growth in software quality and of test progress. This measure can be extended in several ways. For example, the defect detection rate can be contrasted with the correction rate to indicate if debugging is becoming an obstacle to test progress. When an estimate of the total defects to be found is also included, the variance between estimated number of defects and actual corrected defects provides an estimate of progress towards completion. A sample graph is shown in Figure 2-2. Additional measures can be used to focus attention on known problem areas. It is widely agreed, for example, that modified software can be several times more defect prone than new code [Humphrey 1989, pp. 383, 439]. Consequently, in a test effort where frequent changes are required to accommodate evolving requirements, a manager might request data on the relative defect detection rate versus the rate of modification.

Tracking the test progress starts with knowledge of what is to be done and the expected results. In other words, a test plan that specifies the needed testing is a prerequisite for objectively assessing how much testing remains to be performed. Depending on the information provided in the test plan, there are a variety of measures that can be used. When the test plan specifies a required level of structural coverage as a criteria for stopping testing, for example, the number of modules that have successfully reached the required coverage level provides an indicator of early testing progress. When specific test cases are defined, the total tests planned, tests ready to run, tests run, and tests run successfully can be contrasted, as illustrated in Figure 2-3.

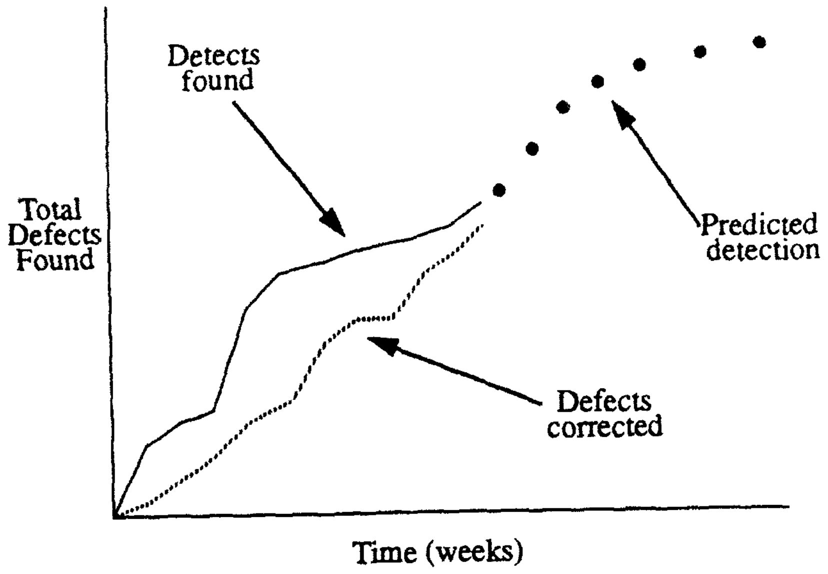

Figure 2-2. Monitoring Defect Detection Rate

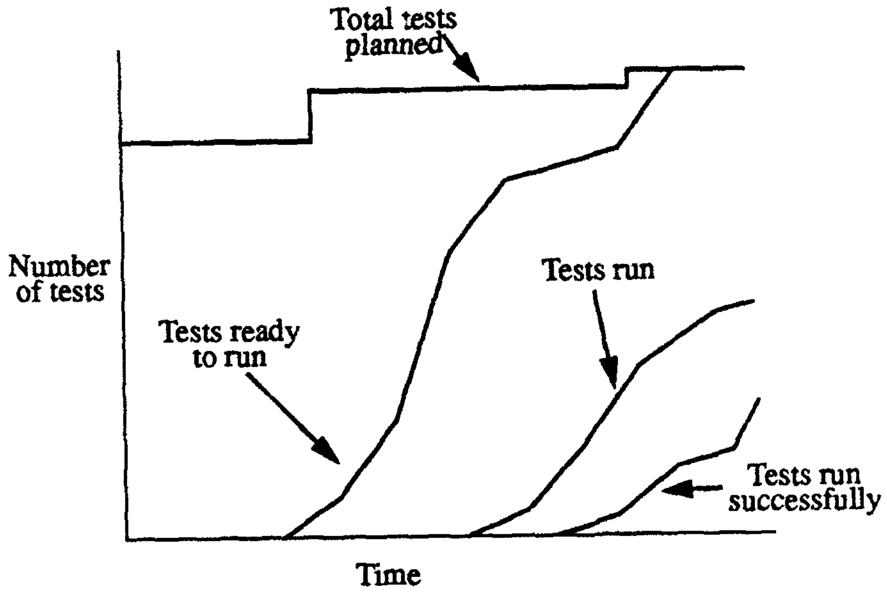

Figure 2-3. Monitoring Test Progress

3. TECHNOLOGY TRANSITION PROJECTS

The goal of technology transition efforts within the testing initiative is to improve current and near-term SDI software testing practices by promoting consistent use of effective tools and methods. Four primary approaches will be used to improve SDI software testing practices:

a. Developing a guidebook that provides SDIO and element contracting agencies with recommended RFP and contract clauses for applying software testing technology.

b. Increasing education and training in desirable testing technology.

c. Promoting software defect prevention efforts within the SDI element programs.

d. Establishing regular forums, such as workshops and Computer Resources Working Group (CRWG) subcommittee meetings, to discuss SDI software testing issues.

3.1 Testing Guidebook

It is the responsibility of SDI element program managers to ensure that reliable element software is cost effectively produced. SDIO should assist element program managers in identifying advanced testing practices that will contribute to reliable, cost-effective software. One method for effecting the consistent application of advanced practices is to require them as part of element RFPs and contracts. This section of the paper discusses a guidebook that recommends RFP and contract clauses for required contractor testing practices and provides rationale for the practices.

3.1.1 Typical RFP Software Testing Requirements

DoD-STD-2167A and DoD-STD-2168 are the primary software development related requirements found in DoD RFPs and contracts. DoD-STD-2168 identifies contractor requirements for developing a software quality management program, but has not had a significant impact on actual software development lifecycle activities. DoD-STD-2167A, on the other hand, plays a key role in the software development process.

DoD-STD-2167A identifies a series of reviews that should take place during software development and suggests a variety of software documentation that should be produced. The standard describes several testing-related documents. The Software Test Plan (STP) describes the contractor's plans for conducting required formal qualification testing (FQT) activities. FQT is the process that allows the contracting agency, or Program Office, to determine whether the software complies with its allocated requirements. The Software Test Description (STD) identifies and describes the specific formal qualification test cases to be applied against each Computer Software Configuration Item (CSCI). The results of the tests are reported in a Software Test Report (STR).

The testing philosophy of DoD-STD-2167A assumes that the contractor knows how to produce defect-free software and that the primary responsibility of the contracting agency is to witness this fact through a series of tests designed to show that the system meets its functional requirements. One problem with this philosophy is that software development is a defect-prone process. Effective use of methods for detecting and preventing defects during software development will result in fewer latent defects upon delivery. Unfortunately, DoD-STD-2167A provides little guidance to program offices and contractors on defect detection and prevention methods. Such methods are rarely required by contracting agencies. As experience is gained with effective methods for defect detection and prevention, contracting agencies should require their use.

The philosophy also assumes that testing is an easily managed, low-priority activity. In fact, testing can be the single most significant cost driver for the development of high-assurance software systems. Consequently, contracting agencies should require proper planning and management of testing activities.

3.1.2 Software Testing RFP and Contract Clauses

Appendix A provides a sample set of suggested RFP and contract clauses that a contracting agency can use to improve visibility into a contractor's software testing process. These clauses place several requirements upon the contractor that are likely to contribute to increased reliability and cost effectiveness of the software. The clauses cover the following areas: formal design and code inspections, structural code coverage, regression testing, testable requirements, automated testing tools, error cause analysis, reliability analysis, and test effectiveness and progress measurement.

The clauses in Appendix A have not been reviewed by contracting offices and those involved with software testing in the SDIO community. They need to be refined and investigation into additional RFP and contract clauses is recommended. For example, in some cases requiring 100% branch coverage may be too stringent. The effort necessary to develop the final 10-15% of test cases may be more effectively spent in other areas. Development of improved RFP and contract clauses should be a high priority in the *software testing initiative*.

Program and contracting offices also need to know why the practices represented in Appendix A should be advocated. Thus, the appendix also provides an explanation of, and rationale for, each clause. Together, the RFP and contract clauses and the rationale form a software testing guidebook that SDIO can use to improve current software testing practices. Active involvement by the contracting agency and the element program office is necessary in order to ensure that the contractor's software testing process is effective.

3.2 Technology Awareness

A key issue for SDIO is "How should advanced testing practices be effectively applied to GPALS software developments?" A similar question is "How should SDIO reduce portions of the technological lag in DoD software testing practice to less than three years?" SDIO cannot afford to wait until appropriate methods are routinely applied by DoD contractors. Thus, SDIO will have to actively foster the transition of advanced testing practices into the SDI contractor community.

It is unlikely that SDI element program offices have sufficient understanding of the testing methods identified in Appendix A to successfully apply them to element development. Unfortunately, these methods are not standard operating procedure for many DoD contractors. Simply placing the clauses found in Appendix A into element RFPs and contracts will not be sufficient to gain benefits of increased reliability and cost effectiveness. Getting these methods successfully implemented by SDI contractors will require efforts by SDIO to raise element program office awareness of the methods. SDIO and its acquisition agents should push element contractors towards the use of key, high payoff testing technologies, such as the rigorous use of inspections, use of structural coverage measures, and the use of commercial tools for test creation, application, and management. This in turn, requires that SDIO and element acquisition officials first understand the need for, and benefits from, these methods. Thus, an important aspect of transitioning these methods will be educating acquisition officials on the technical testing requirements that should be levied on GPALS element contractors. This education should be in the form of brief notices that:

a. Describe the technology and instances of its previous application.

b. Discuss probable benefits and potential pitfalls, ideally with reference to those encountered in previous use.

c. Indicate what to specify in RFP and contract clauses.

d. Describe the contract agency monitoring necessary to ensure the technology is being effectively applied.

These notices should be derived from the contract clause rationale described in Appendix A. However, the notices should be written such that an element program manager will understand the need for, and benefits from, the technology. These notices should be distributed to all SDI acquisition officials that are involved in the procurement of GPALS elements.

3.3 Defect Prevention

Defect prevention is motivated by the high costs of finding and repairing defects. Typically 35-50% of programmer effort is spent in defect removal for sequential software [Dunn 1984, p. 5], and the difficulties presented by concurrent software will necessitate significantly more effort for SDI software. Whereas most types of defect prevention (e.g., prototyping) are only indirectly associated with defects, error cause analysis is a type of defect prevention that is driven by defect occurrence. It will allow element contractors to take information that is already provided by testing activities and use this information to guide improvements to their software development process.

Error cause analysis has been credited with substantial successes. In one case, Hewlett-Packard applied this type of defect prevention in the development of four successive releases of a software product. Whereas 25% of the total defects detected were found after the delivery of the first two releases, this figure dropped to 10% for the third release and zero for the fourth [Humphrey 1989, p. 364]. In another case, error cause analysis provided a 50% reduction in defects found during development and a 78% reduction in defects shipped [Humphrey 1989, p. 365]. For developing shuttle flight software, IBM Federal Systems Division (FSD) Houston combined error cause analysis with software inspections. The resulting improvement in software quality that they experienced over a two-year period is summarized in Table 3-1 [Myers 1988, p. 93].

Error cause analysis requires the analysis of defects to categorize them and identify common defect causes. Action teams are established to devise preventive measures that will avoid recurrences of similar defects, and detection measures that will ensure these defects are identified earlier in the development process. The resulting action items may address development methods, technology, procedures, or training. As with

Table 3-1. IBM FSD Houston Software Quality Improvement

	Industry Defects per KSLOC	**Shuttle Software In 1983 Defects per KSLOC**	**Shuttle Software In 1985 Defects per KSLOC**
Defects detected before release	40 - 85	10	<3
Defects detected after release	8 - 10	2	0.11

software changes, action items are tracked to completion. In this way, error cause analysis provides a continuing focus for process improvement.

The introduction of error cause analysis into the SDI contractor community will incur some costs. Element contractors will need to document their development process. Data on the effectiveness of standard practices should be sought to provide an initial determination of the major problems areas and to assist in setting quality goals. Although problem reports are probably already required, the associated defect classification scheme may need to be revised. (There is a draft Institute for Electrical and Electronics Engineers (IEEE) standard defect classification scheme [IEEE 1987]. However, it is likely that element programs will need to tailor or adapt this scheme.) More importantly, process and product measures will be needed to assess the effectiveness of current practices, as well as the impact of process changes. SDIO should encourage element contractors to use a common, limited set of metrics. Three key metrics used by IBM FSD Houston [Kolkhorst 1988, p. 26] are:

a. Product Defect Rate. For each release, this metric measures the number of defects per thousand lines (excluding comments) of source code (KSLOC). Defects are documented as Discrepancy Reports (DRs):
   Product Defect Rate = Number of valid DRs found post delivery / KSLOC.

b. Process Defect Rate. Computed prior to software delivery, this metric monitors the efficiency of defect detection:
   Process Defect Rate = Number of valid DRs found predelivery / KSLOC.

c. Earl*j* Detection Percentage. Measures the percentage of defects detected prior to integration testing:
   Early Detection Percentage = (Number major inspection defects x 100) / Number of total defects detected.

As with any new technology, error cause analysis and defect prevention skills must be learned and practiced. In addition to the previously cited applications, these metrics can be usefully employed in monitoring the success of defect prevention as a whole.

Despite its proven effectiveness in a few cases, error cause analysis is not widely used. SDIO should seek to gather information on the costs and benefits of this technology that can be provided to element contractors to encourage their contractors to employ this technology. In particular, this information should demonstrate that the investment incurred in setting up and conducting a defect prevention program based on error cause analysis can yield rapid returns [Humphrey 1988, p. 387]. SDIO should also review IBM FSD Houston's education program for error cause analysis and, if appropriate, recommend this training to element contractors.

3.4 SDI Software Testing Workshop

An SDIO forum dedicated to discussing software testing issues does not currently exist. Such issues include, for example, lessons learned from testing methods used in past programs, tools and methods that proved useful, and suggestions for RFP and contract clauses. With SDIO now developing early deployment plans, it is important that the various element program offices and SDIO begin to review and share experience with software testing plans and technology. The formation of a software testing subcommittee of the CRWG would facilitate this exchange. Prior to the formation of a subcommittee, though, it would be useful to bring together the likely participants in a preliminary meeting. The purpose of this meeting would be to promote the application of advanced testing tools and methods. An informal meeting, such as a workshop, is suggested to provide a forum for technical discussions related to software testing. Suggested topics for the workshop include:

a. Element program office software testing plans

b. Element program office positions on needed software testing technology

c. Available software testing methods and tools

d. Experience with metrics related to software testing

e. Software testing clauses for SDI element RFPs and contracts

The workshop should be divided into two parts. The first part would include only those within the SDI community (e.g., element program offices, Army Strategic Defense Command, Air Force Space Systems Division, National Test Facility, SDIO). The second part would be open to others, such as outside experts to give presentations on topics of interest to the SDI community, such as:

a. Commercial testing tool vendor demonstrations

b. Presentations by best-in-class organizations

c. Tutorials by experts in technology areas

3.5 Recommendations

1. **SDIO should ensure that appropriate testing clauses are placed in all element RFPs and contracts.**

   The RFP and contract clauses found in Appendix A are examples of how to address technologies that can significantly improve the reliability and cost effectiveness of SDI software. Without such clauses, and without program office support for their application, SDIO will likely find its software-intensive procurements at higher levels of risk as those found in past DoD software procurements. The clauses in Appendix A have not been reviewed by SDIO or element program offices. SDIO should begin this review process as soon as possible, so that the clauses are ready for inclusion in GPALS RFPs.

   Specifying advanced testing practices in an RFP or contract does not guarantee their proper application. Element program office involvement is critical to the success of these methods. When software is developed using such practices, SDIO should ensure that element program offices are monitoring the effectiveness of these practices and ensuring their appropriate use.

2. **SDIO should query element program offices to determine the potential for the use of defect prevention methods.**

   Defect prevention methods may represent a key driver in reducing software life cycle costs and improving system reliability. Implementation of defect prevention programs is a task best suited to element program offices and their contractors. However, SDIO should provide element program offices and their contractors with information on the potential costs and benefits of defect prevention programs such as error cause analysis. SDIO should also review IBM FSD Houston's education program for error cause analysis and, if appropriate, suggest that element program offices offer this type of training to their contractors.

3. **SDIO should conduct an element program office workshop to promote the application of advanced testing tools and methods.**

SDIO is evolving from a research program to a deployment program. With this evolution, a need exists for regular communication among element program managers and SDIO on the activities of software testing. However, element program offices are currently placing little emphasis on the processes and technology for testing SDI element software. A workshop would provide a useful forum to discuss element program office test plans, RFP and contract clauses, and so forth.

4. **SDIO should establish and actively participate in a software testing subcommittee within the CRWG.**

The CRWG provides a regular forum for discussing a wide variety of SDI computer- and software-related topics. A software testing subcommittee should function as an advisory group for element software testing activities. Members of this subcommittee should have direct responsibilities for element testing activities. Suggested activities for this group include the review of element software testing plans and results of testing activities, discussion of available software testing tools, and discussion on lessons learned from application of testing technologies.

The workshop previously described would provide a useful indicator of the need and timeliness of a CRWG software testing subcommittee.

4. TECHNOLOGY EVALUATION EXPERIMENTS

The primary goal of software testing technology evaluation experiments is to identify promising new or existing testing techniques that can add significantly to the trustworthiness, reliability, and quality of SDI software. Of course, software testing must also be cost-effective and affordable. New testing approaches and techniques are being developed continuously to address software development problems. Many techniques have been convincingly demonstrated in research environments but need improvement and refinement before they can be broadly applied. Software developers and program managers, understandably, are reluctant to absorb the risk of experimenting with promising but unproven technology. The technology evaluation experiments, therefore, are intended to:

a. Identify from the collection of available software testing techniques those that offer the greatest promise in reliably and cost-effectively detecting the most critical defects.

b. Bridge the gap between research and routine practice, and accelerate the transition of useful testing techniques.

c. Absorb the costs and mitigate the risks of applying new, unproven testing techniques.

d. Expedite feedback to researchers and technology developers on necessary improvements and refinements.

The principal question that should be addressed by the participants in an experiment to evaluate a particular testing technique is "To what extent can the use of this testing technique increase the level of assurance achieved in software we develop, and at what cost?"

This can be subdivided into the following specific questions:

a. What measures of improved trustworthiness, reliability, or quality can be shown? For example, higher success rates in detecting defects, detecting different types of defects, and detecting defects earlier.

b. What costs and cost savings are associated with the use of this particular technique? For example, training and tools required, time to apply the technique or time saved, and run-time performance impact.

c. Are there any potential pitfalls that need to be avoided? For example, incomplete or inadequate software specifications.

d. Are there any changes to the tools or methodologies that would improve their effectiveness in the SDI environment?

e. How could this technique be used most cost effectively in conjunction with other techniques?

4.1 General Approach

The most effective approach to achieving these goals is to:

a. Establish the spending authority and a stable budget for software testing technology evaluation experiments.

b. Have SDI software developers, working through their program offices, propose specific experiments to evaluate testing techniques that they believe have the potential to significantly improve the products they are developing.

c. Fund the most promising experiment proposals.

d. Plan to incorporate positive experimental results in the technology transition activities described in Section 4.

Participation by SDI software developers and program offices in these experiments is essential to moving proven testing techniques into standard practice within the SDI community.

4.2 Experiment Scenarios

There are two general approaches for testing technology evaluation experiments. The preferred approach is to apply a new or existing testing technique in the course of developing non-critical path segments or components within ongoing, "live" software development projects. The alternate approach is to create parallel development projects, isolated from those producing scheduled deliverables, or separate software projects created explicitly for experimentation.

4.2.1 Live Project Scenario

A hypothetical live project testing experiment may provide a useful illustration. The Brilliant Pebbles Software Development Plan identifies concurrent software and hardware development at multiple sites as risk factors [BP-SDP 1991, p. 35]. Such development requires close coordination of interfaces, which can be enhanced by rigorous specifications. The contractor would likely use Ada package interface specifications, but this facility does not provide sufficient confidence in interface behavior. As an experiment to improve interface coordination, the contractor proposes (hypothetically) to develop formal protocol specifications for the required behavior across the interfaces, and from these to generate simulations of either side of each interface for testing. Since this is new technology with limited tools and staff experience, initial development of protocol specifications would be restricted to a small number of software components. The selected components, however, would be pieces of deliverable, operational CSCIs.

Applying unproven technology within live projects involves some risk. If developers are accountable for this risk, very little experimentation will occur and technology transition will be slowed. When a developer takes the risk of experimenting on his own and gets "burned", getting him to try other new techniques will be very difficult—no matter how successful these other techniques are. In addition, he is likely to reject the technique that caused him so much trouble even after it matures and is proven to be effective. One of the key objectives of specifically sanctioning experimentation is that the risk of new technology is recognized and absorbed at a higher level. Moreover, the risk is reduced, however, if:

a. A non-critical-path component or subsystem within the project can be identified as the subject for the experiment. That is:

   (1) The project's schedule can absorb a potential stretching of that component or subsystem's completion schedule, and

   (2) Additional man-power or expertise can be applied to overcome difficulties that might threaten that component or subsystem's timely completion.

b. The new testing technique or approach matches the characteristics of that component or subsystem (for example, real-time testing techniques for components with critical timing requirements).

Two principal advantages of live project experimentation are that new techniques are evaluated under actual operational conditions and that successful results can be transferred more readily into routine practice. If a new technique is successful in a live experiment, programmers working on other components and subsystems will learn about it and want to use it to improve their work. If product quality is improved and the fears of potential cost increases or schedule impacts are dispelled, managers of other projects will want their programmers to use it. Determining that a particular technology does not provide anticipated benefits or that it introduces unforeseen problems is also valuable information that needs to be fed back into the research program. It does not mean that the experiments or experimenters were unsuccessful.

4.2.2 Parallel Development Project Scenario

The parallel development approach to technology experimentation sets up a separate experimental project in parallel with the actual, live development effort. The parallel project takes the same requirements and specifications as the live development effort but, instead of (or in addition to) using the standard technologies applied in the live effort, the parallel project applies a selected experimental technology. An example of this approach is the experimental evaluation of "ANNA" currently planned at the National Test Facility on the Level-2 System Simulator. ANNA's formal program annotations will be developed for portions of the simulator and the ANNA tools will be used to augment the testing of this code.

A significant advantage of the parallel development approach is that it separates the risks of the experimental technology from the live system development effort. The application domain for the experiment is identical to the live system. The experiment team need not work to the same deadline schedule as the live system development, which allows time for learning how to apply the new technology and how to operate new prototype tools. The only impact that such experimentation might have on the live development effort is that it may divert a number of competent staff members from the live system effort.

4.2.3 Separate Evaluation Project Scenario

Technology experimentation can also be accomplished by setting up separate projects to develop similar or related software. That is, these projects do not have to respond to the requirements and specifications of a particular live system development effort. This approach offers much more flexibility in arranging and conducting experiments because exact specifications of live system requirements are not necessary. Experiments can proceed when the technology is ready and need not wait for the details of a particular application for evaluation. The development specifications for the experiment, however, must be sufficiently realistic to determine the value of the technology in actual live system development conditions.

4.3 Participants

Participants in testing technology evaluation experiments should be drawn from all SDIO program offices and contractors involved in software development. In particular, this includes the National Test Bed and Nation:¹ Test Facility (NTF) offices, the Army's Strategic Defense Command, the Air Force's Space Systems Command, and their contractors. There is ample opportunity to conduct significant testing experiments within the projects managed by these offices. Successful results from experiments conducted by these groups can be applied immediately on a wider scale and turned into revised standard testing practices.

A second tier of participants who may be able to evaluate less mature but potentially high-payoff testing techniques include the Service laboratories, the National laboratories, and DoD and other government research agencies.

4.4 Technology Selection Criteria

Any given list of specific candidate technologies would probably be incomplete and would rapidly become obsolete. It might also be interpreted to mean that evaluations of other testing methods or of new developments in testing techniques were arbitrarily restricted. Rather than recommending specific technologies, therefore, we established the following criteria to guide prospective experimenters in selecting technologies for evaluation:

a. Criticality and applicability—To what extent does the technology address anticipated testing deficiencies in the particular software project? For each critical software characteristic,[3] a high rating implies the technology directly addresses these deficiences, a medium rating implies indirect or partial applicability, and a low rating implies little or no applicability.

b. Maturity and practicality of the theory—To what extent has the theory behind a technology matured toward practical application? For maturity, a high rating implies a well-developed theory, a medium rating implies an incompletely developed theory, and a low rating implies an undeveloped theory. For practicality, a high rating implies successful practical use, a medium rating implies demonstrated laboratory use, and a low rating implies little or no demonstrated practicality.

3. Critical SDI software characteristics that pose significant problems for achieving high levels of assurance include concurrent, distributed (space and ground), fault tolerant, large scale, and real time.

c. Availability and maturity of tools—To what extent have tools been developed to support the practical application of a technology? A high rating implies existence of robust, commercially available tools. A medium rating implies existence of complete prototype tools that are in use by people other than the developers. A low rating implies incomplete prototype tools that are not generally available from developers.

d. Feasibility—To what extent is the technology feasible for use on a particular project in terms of cost, ease of use, and project time frame. A high rating implies low cost, minimal training, and workable time frame. A medium rating implies moderate cost, moderate training, or tight time frame. A low rating implies high cost, extensive training, or unreasonable time frame.

e. Potential benefit—To what extent is the technology likely to benefit this and other SDI software efforts, including full-scale development? This is a composite rating that considers the effect of overlapping technologies. A high rating implies a significant contribution to software assurance not covered by other technologies at lower cost. A medium rating implies a respectable contribution not adequately covered by higher rated technologies. A low rating implies a relatively small contribution, but one that is not adequately covered by higher rated technologies. A rating of "covered" implies that a technology's contributions are adequately covered by higher-rated or lower-cost technologies.

4.5 Example Technologies

To give examples of how the selection criteria above are intended to be used, several candidate evaluation technologies were rated. In each of the subsections below the particular testing technology is briefly described and these ratings are listed along with a short rationale.

4.5.1 ANNA

ANNA provides a means for detecting software defects that violate assumptions about the intended behavior of software during execution. Its mechanism for detecting these violations provides a type of self-checking that can also be used to support fault tolerance in addition to software testing. ANNA is an extension of Ada that adds facilities for specifying these assumptions. The extensions are called "annotations" or "assertions", and are based on techniques used for formal program specification. Annotations can be viewed as a formal style of embedded program documentation. In fact, without the ANNA tools, annotations are Ada comments.

Simple ANNA annotations allow software developers to specify, in a formal way, relationships among variables that should hold at various points within a program. More sophisticated annotations allow similar specifications for subprogram input-output relationships, conditions that should hold throughout the program for a particular variable or for all values of a particular type, conditions surrounding exceptions and exception handling, and relationships among generic parameters.

Annotations can be used early in the software development process to capture concrete specifications and design intentions. Often, important relationships among inputs, outputs, and internal data can be easily stated, even though producing those outputs may be extremely complex. ANNA provides a mechanism for specifying those relationships without having to develop the code that will compute the results.

Later, when the code is developed, the same annotations can be used in testing. The principal ANNA tool is a translator that converts annotations into executable Ada code that verifies the specified conditions during program execution. If one of these conditions is violated at any point during execution, the violation is reported and an exception is raised. Annotation violations often reveal defects long before their effects appear in a program's output. Several excellent descriptions on the use of annotations in program testing are available [Luckham 1990, 1984]. One implementation and various ANNA-related papers are available on the Internet host "anna.stanford.edu" (see the file "pub/anna/read.me" for more details).

The five technology selection criteria for ANNA are ranked in Table 4-1.

Table 4-1. Ranked Technology Selection Criteria for ANNA

Criticality & Applicability	High (with limitations for real-time components)
Maturity & Practicality	High
Tool Availability	Medium/High
Feasibility	High (project dependent)
Potential Benefit	High

Useful annotations can be added to virtually any Ada code. The only limitation is that checking specified conditions at run-time consumes time (and, to a lesser extent, memory), which may affect time-dependent computations. ANNA is based on well-developed, practical theory. Although not commercial products, the tools available from Stanford University are complete and robust. ANNA should be easy to incorporate in a wide range of development projects. ANNA's formalization of a program's required behavior and its ability to expose internal defects before their effects appear in output significantly increase confidence in the program's correct operation.

4.5.2 Cleanroom

Cleanroom was developed at IBM and represents many years of research and experimentation. Cleanroom is actually a complete software development methodology, not merely a testing technique. Testing is treated much differently in Cleanroom than in most other development approaches. Experiments with Cleanroom need not be limited to evaluating its approach to testing, however, since other aspects of the methodology also contribute to software quality.

At the outset, Cleanroom requires development of formal function and performance specifications. The system must be specified in terms of its required mathematical functionality. A design is then developed based on a state machine model of algorithms to compute the required functions. The design is verified against the formal specifications by rigorous logical arguments. Code is developed by teams in stages of roughly ten-thousand line increments. The first stage is an executable system superstructure and each additional increment adds new functionality to the system. Code is verified against the design and the formal specifications by rigorous logical arguments.

In the strictest interpretation of the Cleanroom methodology, no unit-level testing or debugging by code developers is allowed.[4] A separate, independent team of testers is responsible for generating and executing all tests. Tests are based on the formal system specifications and are developed concurrently with design and code development. Testers do not use information about the design or code structure in developing the tests. Design and code development teams do collaborate with testing teams, however, to assure that the requirements are accurately and unambi uously specified. Test results and discrepancies are returned to the development team for any necessary corrections. Complete regression testing of existing functionality is performed as each developmental increment is added to the system.

4. The objective of this rule is to avoid hasty code modifications that can introduce new detects almost as quickly as the original ones are corrected, when design and coding decisions need to be reconsidered. IBM has found that conventional unit testing and debugging by programmers is not the most effective use of their time.

Cleanroom also seeks to provide statistical measures of the system's reliability, such as its mean time to failure (MTTF). To determine these measures, test data are generated *randomly* based on profiles of expected system use. The intended effect of using these profiles is that frequently used functions are tested more thoroughly than seldomly used functions. The number of needed tests is driven by the level of reliability required and the number of defects discovered. In addition to reducing the frequency of defects users are expected to encounter (as opposed to the number of delivered defects), this form of statistical quality control allows management to track quality and productivity trends.

The five technology selection criteria for Cleanroom are ranked in Table 4-2.

Table 4-2. Ranked Technology Selection Criteria for Cleanroom

Criticality & Applicability	High
Maturity & Practicality	High
Tool Availability	Low/Medium
Feasibility	Medium (project dependent)
Potential Benefit	High

Cleanroom techniques can be applied to any software development project. It is based on well-developed, practical theory. Many aspects of the Cleanroom approach could be supported by tools. No integrated collection of tools has been established to support Cleanroom, although many tools are applicable. Several aspects of the Cleanroom methodology require a moderate amount of training but otherwise it should be relatively easy to incorporate in a wide range of development projects. Cleanroom's formal, incremental development approach and its unique approach to testing offer significant potential advantages in building systems efficiently and with measured reliability.

4.5.3 Improved Structural Coverage

Structural coverage enables developers to track which parts of their software have been exercised by various tests. The process usually involves a code instrumenting tool, which alters the program code to cause trace data to be written to a file, and a coverage analysis tool, which reports execution results in meaningful ways. Examples of the statistics that are commonly collected include the number of times each statement is executed, the number of true and false results at each conditional branch point, the number of iterations of each loop, and the number of times each subprogram is called. Most coverage analyzers can combine the statistics from several test runs to give cumulative test coverage results.

Ad hoc testing techniques, without the aid of coverage analysis, typically achieve coverage of only about 50 to 60% of the code in complex systems. That is, in common testing practice as much as half of the code in a system is not even executed. Complete, 100% coverage may be extremely difficult to achieve in final, system-level tests. Lower-level unit tests, however, should guarantee that every possible statement and branch in the code have been exercised. Without some form of coverage analysis, developers have little information about how well their products have been tested. This is the reason for the coverage testing requirement recommended in Appendix A.

Although basic coverage testing techniques are mature and well-supported by automated tools, there are several areas where current research is working toward improvements. Any of these areas could form the subject of an evaluation experiment. They include the relative cost-effectiveness of:

a. Different types and different degrees of structural coverage (e.g., data flow and linear code sequence and jump (LCSAJ)).

b. Automated support for deriving test data that will exercise specific branches or code segments.

c. Non-invasive monitors for real-time software, which do not require code instrumentation or alter runtime behavior.

The five technology selection criteria for improved structural coverage techniques are ranked in Table 4-3.

Table 4-3. Ranked Technology Selection Criteria for Improved Structural Coverage

Criticality & Applicability	High (with limitations for real-time components)
Maturity & Practicality	Medium/High
Tool Availability	Medium/High
Feasibility	High
Potential Benefit	High

Test coverage data can be collected on virtually any Ada code. The only limitation is that instrumented code consumes additional time and memory during execution, which may affect time-dependent computations. Test coverage data collection and analysis are based on well-developed, practical theory. There are several commercial tools that support code instrumentation and coverage reporting beyond branch coverage, as well as non-invasive monitoring tools. Improved coverage testing should be easy to incorporate in a wide range of development projects. Improved coverage testing techniques and non-invasive monitoring tools can contribute significantly to the assurance of correct, reliable system operation.

4.5.4 Reproducible and Deterministic Execution

Reproducible program execution is an added complication for testing introduced by concurrency, whether the code is executed on a single central processor, on tightly coupled parallel processors, or distributed over multiple loosely coupled processors. Whereas sequential program execution can be repeated by setting up the same input conditions, concurrent program executions may behave differently even with the same inputs. This is because nondeterministic operations may produce different results. For example, the relative progress of concurrent processes is generally unpredictable. Correct programs produce correct results in spite of these possible variations. Incorrect programs, however, cannot always be forced to produce incorrect results, which may create a false sense of confidence in the program's behavior.

Concurrent processes communicate through a series of synchronization events. The exact sequence of synchronization events is what determines the program's behavior. For a given set of program inputs, several different synchronization sequences may be possible, each one potentially producing different results. Developers and testers of concurrent software, therefore, must be aware of all the possible synchronization sequences possible within the program. For all but the simplest programs, this requires automated analysis.

Reproducible testing requires repeating exactly a particular synchronization sequence. One approach is to transform the concurrent program into a sequential program that contains a fixed interleaving of the original concurrent operations and effectively "hard codes" the one synchronization sequence. A refinement of this technique, called deterministic execution, can be achieved by a less drastic program transformation and allows concurrent execution of parts of the code that cannot distort the results.

The five technology selection criteria for reproducible and deterministic execution are ranked in Table 4-4.

Table 4-4. Ranked Technology Selection Criteria for Reproducible and Deterministic Execution

Criticality & Applicability	High (for concurrent systems)
Maturity & Practicality	Medium
Tool Availability	Low
Feasibility	Medium (project dependent)
Potential Benefit	High

Reproducible and deterministic execution testing is applicable to all concurrent software systems. The theory behind these techniques is reasonably well developed; however, its practicality has not been demonstrated beyond relatively small-scale laboratory use. A number of concurrent system testing and debugging tools have been described in the literature, but few of them appear to be readily available outside their development laboratories. Experiments with reproducible and deterministic execution testing would likely incur moderate tool installation and maintenance costs, as well as moderate training costs. These techniques promise significant advantages for testing concurrent systems because conventional testing techniques provide essentially no support in this critical area.

4.5.5 Software Fault-Tree Analysis

System safety engineering is a process of identifying hazards, assessing the risks they represent, and then designing safeguards that eliminate the hazards or reduce the risk factors to acceptable levels. A "hazard" in this context is any set of conditions where unacceptable damage, injury, or other loss is possible. The "risk" associated with a hazard is directly related to the likelihood of the hazard occurring, the likelihood that the hazard will lead to an accident, and the potential loss or cost incurred by such an accident. It should be clear that malfunctioning software in weapons control systems can easily contribute to high-risk, hazardous conditions.

Fault-tree analysis is a technique of analyzing systems for conditions that could lead to hazardous system states. This technique traces potential accidents back through the events and conditions that would enable them to occur. Once hazards are recognized, the system can be designed to avoid them or to include safety devices such as pressure relief valves, electrical interlocks, and warning indicators.

Fault-tree analysis can also be applied to software and is particularly valuable for software in embedded computer systems. Sources of software defects that can contribute to hazards include coding mistakes, design flaws, and defects in requirements and specifications. System developers must first verify that software requirements are consistent with the system's safety criteria. This is essential so that the software developers can address in their designs any critical conditions or events that the software must either avoid or ensure. System hazards can then be mapped into corresponding unsafe software outputs and the software can be analyzed to determine under what conditions unsafe output might be produced. In addition to avoiding certain hazards the design can include run-time software safety checks. These checks generally require little processing time but can add significantly to overall system safety.

The five technology selection criteria for software fault-tree analysis are ranked in Table 4-5.

Table 4-5. Ranked Technology Selection Criteria for Software Fault-Tree Analysis

Criticality & Applicability	High (for fault-tolerant and safety-critical systems)
Maturity & Practicality	High
Tool Availability	Low
Feasibility	High
Potential Benefit	High

Software fault-tree analysis is generally applicable and of particular value in embedded real-time computer systems. It is based on well-developed, practical theory. At present, software fault-tree analysis is primarily a manual process, although prototype tools are being developed. With improvements in tool availability, software fault-tree analysis should be easy to incorporate in a wide range of embedded system development projects. Avoiding recognized hazards or detecting and responding to unsafe conditions at run-time significantly increases confidence in a system's safe operation.

4.6 Recommendations

1. **SDIO should annually budget for several software testing experiments to identify promising new testing techniques.**

Historically, SDIO has not provided funds for software testing experimentation. In fiscal year 1992, funds were allocated for an initial NTF experiment with ANNA. Such efforts should be encouraged, and it is hoped that additional experimentation will occur. However, the scope of experimentation must be broadened and expanded.

SDIO should solicit specific testing experiment proposals from development organizations (e.g., Army Strategic Defense Command, Air Force Space Systems Division) that play key roles in GPALS software development. The proposals need significant input from the contractors, so the solicitations need to filter down to them. SDIO should also coordinate with Service research and development agencies, such as the Naval Research Laboratory and Rome Laboratory, to identify promising avenues of experimentation.

2. **SDIO should provide recognition for innovations in software testing that evolve from these experiments.**

Achieving the necessary levels of assured software performance and reliability will require identifying and applying new solutions to software testing problems. One of the fastest ways to advance the current state of testing practice is to reward innovative application of new, but unproven, tools and techniques. Software testing activities are sometimes viewed as counterproductive by development teams. As software development deadlines approach, improved testing can mean that more corrections and rework are required. Program offices, however, must take a broader view and recognize that improved testing leads to improved products. Rewards for innovation in testing may include letters of commendation, presentations to the SDI community, and notice in internal publications.

3. **SDIO should plan to incorporate positive experimental results in technology transition activities.**

The goal of these experiments is to identify technology that can improve current software testing capabilities. As positive experiment results appear, SDIO should ensure that the results are communicated to the SDI software development community. Avenues for this communication include presentations at CRWG meetings, updates to established software testing standards, and revisions to RFP and contract Statement of Work requirements for software testing practices.

5. RESEARCH PROJECTS

The goal of the research part of the testing initiative is to address several fundamental problems of testing large-scale, concurrent, distributed, real-time, and fault-tolerant software systems. Current methods and techniques for testing such systems cannot provide the level of assurance of correct operation, safety, and reliability necessary for SDI software. Without solutions to these problems, SDI software is likely to contain latent defects that could compromise the success of SDI's mission.

The general approach of this research program includes:

a. Strengthening the scientific basis for software testing methods, tools, and metrics. Many commonly used testing approaches are ad hoc and produce variable results. Automated tools help make testing easier and more systematic, but the technology underlying current tools lacks sufficient basis for predicting or drawing other conclusions about a system's correct operation, safety, and reliability.

b. Scaling up testing capabilities to rigorously test large, complex software systems. Because of their complexity, interactions among components of large systems cannot be tested exhaustively. It is relatively easy to demonstrate, however, that partial application of current testing techniques does not achieve sufficiently high levels of confidence.

c. Promoting collaborative efforts between researchers and system developers. The most effective way to direct the necessary research and to transform research results into practical testing techniques is to combine the interests and efforts of the research and system development communities.

This research program must focus on developing and extending innovative approaches to software testing that have the potential to significantly advance the state of the art and practice. That is, the research should be directed toward breakthroughs that will enable rigorous, systematic, and repeatable testing of software systems. Incremental improvements in current testing techniques that reduce the cost of testing or extend the range of software characteristics that can be tested are also expected. Incremental improvements alone, however, are not likely to achieve the significant results required for assuring the correct operation, safety, and reliability of SDI systems.

Specific areas requiring attention include:

a. Methods for testing highly reliable, real-time, concurrent, and fault-tolerant software. These attributes characterize many critical software components in SDI elements. Software with any one of these attributes is difficult to test, and they usually occur together. Current techniques cannot adequately test software that combines all of these attributes as SDI software will.

b. Methods that scale up for testing very large software systems containing millions of lines of code. Small programs can be tested relatively easily. The same testing methods and techniques, however, do not scale up to large-scale software systems. Testing large-scale SDI systems will require more than simply expanding the number of small-scale tests.

c. Methods that scale up for testing software that may be distributed across systems containing potentially hundreds of processors. Distributed software systems are time sensitive, concurrent, and fault tolerant. In addition, there may be significant latencies in communications among components and strict synchronization of concurrent activities is impossible. This means that testing must be accomplished without complete control over the system under test and without complete knowledge of the system's state. Ad hoc techniques that have been used to test small-scale distributed systems cannot be relied upon to meet SDI's needs for assuring large systems.

d. Methods for evaluating and testing software system requirements and designs. The cost of correcting defects in software grows exponentially as the time between their introduction and detection increases. In particular, requirements and design defects that are not discovered until integration and system testing often cause significant budget and schedule overruns. Formal methods, for example, may prove useful in rigorously specifying requirements and designs. The focus of these techniques, however, has to change from proving the correctness of small-scale programs to modeling and prototyping large-scale systems.

5.1 Anticipated Benefits

The anticipated benefits of results produced from this research will help assure the correct operation, safety, and reliability of SDI software. Specifically, this will be achieved by developing new testing methods and techniques that will enable early, systematic detection of latent software defects that currently cannot be reliably detected.

Further benefits include increased understanding of software defects and how they can be prevented and detected, increased visibility into software reliability throughout the development process, and reduction of the cost and time required to achieve specified levels of confidence in the correct operation, safety, and reliability of complex software systems.

5.2 Evaluation Criteria

Research proposals and the progress of on-going research projects must be scrutinized to ensure that the most promising, high-payoff ideas are supported. The following criteria are recommended for evaluating proposed research projects:

a. Relevance, utility, and potential impact of the research. Software developers are on the front lines facing the difficulties of adequately testing software. They know where the problems lie and, therefore, should be consulted, along with testing experts, in evaluating the relevance and potential impact of proposed research.

b. Innovative character and intrinsic merit of the research. This will require evaluation of proposals and progress papers by recognized experts in the software testing field. Several reviews of proposals and progress papers should be sought to ensure balanced evaluations.

c. Costs and anticipated lead time for results that could be applied and evaluated in SDI software. Although this program is intended to address the more fundamental research issues in software testing, costs and lead time must be considered in selecting proposals.

d. Qualifications, competence, and productivity of the research team. Results of previous research are good indicators of competence and productivity. Highly relevant and innovative ideas from new groups or individuals, however, should also be considered along with good ideas from researchers with successful track records.

e. Collaboration with and cost sharing by industrial partners. Active participation in research programs by companies with substantial software business bases is often a good indicator of promising ideas. It also helps focus research to ensure relevant and useful results. Such companies often have experience moving research results quickly from the laboratory into practice. Cooperative funding of research by government and the private sector, therefore, should be encouraged.

5.3 Participants

There are three major categories of participants in the research program:

a. Research investigators

b. Evaluators of research results

c. Research funding agencies

At present there are relatively few recognized centers of academic and industrial research that focus on software testing. One approach to stimulating innovative software testing research is to expand this research base. Universities are expected to provide most of the primary investigators from their faculties and graduate student populations. Industry, military, and other government software laboratories may provide additional researchers.

Experimental evaluation of new testing techniques is a key part of the testing initiative. Industry, military, and other government software laboratories, in particular the NTF and Naval Research Laboratory (NRL), are expected to play important roles in these evaluations. SDI element development sites offer additional opportunities to evaluate prototypes of testing tools as they emerge from the research program.

Military and government research agencies such as the Defense Advanced Research Projects Agency (DARPA), Office of Naval Research (ONR), Rome Laboratory, and the National Science Foundation (NSF) provide natural channels for funding and administering software testing research. Both NSF and DARPA are considering plans to sponsor a basic research program in software testing. The Navy is sponsoring a dependable computing initiative and a real-time software initiative through ONR, and a large research program in engineering complex systems through the Naval Surface Weapons Center. SDI's resources could be joined with these programs to gain additional leverage.

5.4 Recommendations

1. **SDIO should establish and fund a long-term research program to improve testing capabilities for large-scale, concurrent, distributed, real-time, and fault-tolerant software systems.**

A sustained, long-term research program would likely need on the order of $1 million per year for five years. This amount is likely to produce useful results within the available time frame. Given the current shortage of active testing research investigations, it was judged that much more than this amount could not be applied productively toward SDI's needs. To revitalize an active software testing research community it is important that this program is recognized as a sustained, multi-year effort. Stop-and-start funding will defeat program objectives. Also spreading the money thinly over too many intermediate funding agencies could dilute the program's impact. By taking a leadership position in establishing this program, SDIO can expect other research funding agencies to cooperate and perhaps even contribute additional resources of their own to extend the program.

2. **Once the research program is initiated, SDIO should closely monitor the progress and effectiveness of research projects. Methods for advanced experimentation and evaluation of research results should be developed.**

Annual reviews of ongoing research projects should be conducted. Renewed funding should be contingent on demonstrated progress and on the expectation of useful results. In addition, annual workshops should be held where both research ideas and practical problems of testing SDI software can be presented and discussed. This mix of theory and practice should provide two-way communication between researcher and practitioner, accelerating the transition of new research results into practice and feeding practical utility and relevance information back into the research process.

APPENDIX A

SUGGESTED RFP AND CONTRACT CLAUSES

This appendix contains draft Request for Proposal (RFP) and contract clauses suggested for use in SDI software procurements. Rationale is provided for each of the clauses, and the clauses are identified by indented margins and single-spaced text. DoD-STD-2167A requires a Software Development Plan (SDP) to be included as part of an RFP response. The requirements specified below refer to sections of this plan as described in Data Item Description (DID) DI-MCCR-80030A. In addition, the following definitions are used:

> **Operational Software**: includes all software necessary to operate or control any system of which it is a part.
>
> **Support Software**: includes all ancillary software developed under this contract to produce, configure, test, install, or maintain operational software.

A.1 Inspections

Inspections have been found to be two to four times more cost-effective in detecting code defects than execution-based testing [Russell 1991]. Furthermore, inspections often find up to 80% of all code defects. An operating system development organization found that design defects were found in one sixth the time by inspections [Ackerman 1989]. Inspections are not only more cost effective than many execution-based testing, they can also find defects that are difficult to find using execution testing techniques. Although the practice of inspections will require a major commitment of time and funds, their proper application will reduce overall system development costs. Additional evidence of the value of inspections can be found in the literature [Fagan 1986, Humphrey 1989, Weinberg 1984].

Inspections are consistent with DoD policy. DoD Instruction 5000.2 states that "walk-throughs, inspections, or reviews of requirements documents, design, and code . . . should be used." Formal inspections differ from walk-throughs and informal reviews, however. In particular:

a. Formal inspections require that specific roles be filled and that certain procedures be followed. Key roles include those of inspection moderator, author, reader or paraphraser, and recorder or scribe. Training for all participants in the objectives and procedures is recommended. Inspection moderators should have had special training in how to conduct inspections. Procedures include use of checklists to ensure that thorough checks are made for specific, frequently occurring types of defects.

b. Management is usually excluded from inspections. The objective of inspections is to expose technical problems, not to evaluate employee performance. Without management present, reviewers tend to find more defects. These defects are also less expensive to correct because they are discovered earlier in the development process.

c. Formal inspections cover small segments of requirements, designs, or code in great detail. Inspection effectiveness has been found to be directly related to the rate that material is covered [Russell 1991, Fagan 1986]. Attempting to cover more than four to five pages of detailed text or code in a two-hour inspection significantly reduces inspection effectiveness. Inspection meetings that last longer than about two hours, inspections scheduled back-to-back, and participating in more than two inspections per day all reduce effectiveness.

d. Formal inspections require a small number of participants. The optimal working group size for inspections is on the order of five to eight people. Larger groups require more organization to convene and uncover fewer problems.

e. Formal inspections require preparation on the part of reviewers. Reviewers may spend on the order of three to five hours studying work products and related material to prepare for each review. Russell reports that a useful rule-of-thumb for estimates of code inspection resources in elapsed time is three times the number of thousand-lines of code inspected [Russell 1991]. While this may seem like a great deal of time, it is one half to one sixth the time necessary to find and fix program defects later in the development process.

Like all other forms of testing, inspections alone cannot reveal all defects, so while inspections provide a cost-effective method for revealing a large proportion of defects, execution-based testing is still necessary.

Suggested clauses for inspections:

All operational and support software designs and code shall undergo formal inspections, similar to those described in [Fagan 1976] and [Fagan 1986]. Proposals shall describe, in Section 4.2.1 of the SDP, how formal design and code inspections will be implemented, and shall include proposed inspection checklists for both design and code. In addition, the following process documentation is required:

a. During software design, the contractor shall provide monthly summary statistics of the design inspection process. This summary shall include the number of design units inspected, along with the number, type (as denoted in the checklist), cause, and resolution of defects detected.

b. Prior to the Critical Design Review (CDR) a signed statement shall be provided verifying that the designs for each Computer Software Configuration Item (CSCI) have been fully inspected.

c. During software coding, the contractor shall provide monthly summary statistics of the code inspection process. This summary shall include the number of Computer Software Units (CSUs) inspected, along with the number, type (as denoted in the checklist), cause, and resolution of defects detected.

d. Prior to the beginning of CSCI integration, a signed statement shall be provided verifying that all code units have undergone formal inspection.

A.2 Structural Test Coverage

Structural testing requires that every instruction or expression of a program be executed or evaluated at least once during testing. Structural coverage does not imply correctness but is a confidence raising technique. The fact that each statement, say, has been executed does not mean the statement is correct. We can, however, have more confidence in that statement than one that has never been executed during testing. Clearly, if an instruction has never been exercised by any test, there is no evidence that this instruction will work correctly when executed in operation.

There are several levels of structural testing. The simplest is statement coverage, which requires every statement to be executed at least once by some test. Multiple tests are almost always necessary to achieve 100% coverage. Little confidence can be gained from statement coverage, because very few logic flows are exercised during this type of testing. The next level is branch coverage, which requires every possible branch direction to be traversed at each decision point. Branch coverage encompasses statement coverage. Higher levels of coverage require more complex combinations of code be exercised, and may therefore require more test cases to achieve full coverage.

Complete branch coverage is feasible at the unit (subprogram) level. There are many commercial off-the-shelf (COTS) tools available, such as test harnesses, that support branch coverage testing [Sittenauer 1991, Youngblut 1991]. As software components are assembled and integrated into subsystems, controlling the execution of individual statements and branches within low-level components becomes successively more difficult. For large systems with many components and subsystems, full structural coverage testing of the entire system—top to bottom—is often impractical. In this case, a stratified approach that tests components and subsystems separately should prove adequate. That is, as tested components are assembled into larger subsystems, only the upper layers of program structure need to be tested to satisfy the coverage requirement.

Complete structural coverage does not guarantee that a program will always execute correctly. A program can succeed in some cases (several of which may have been tested) yet fail in others, under different conditions. Still, this type of testing generally has the highest defect yield of all execution testing techniques [Humphrey 1989, p. 196].

Structural test coverage is consistent with DoD policy. DoD Instruction 5000.2 states that "rigorous testing of modules and interfaces at all levels of aggregation . . . should be used." Other organizations also believe it is practical and already require such coverage. For example, the United Kingdom Ministry of Defense requires complete (100%) branch testing in its interim Defense Standard 00-55 (Section 33.2) [MoD 1991, p. 19].

Suggested clauses for structural test coverage:

In addition to the functional testing required by DoD-STD-2167A, all operational software shall be tested to achieve the following levels of structural test coverage:

a. All statements.

b. All branches for both true and false conditions, and case statements for each possibility, including "otherwise" (e.g., the Ada "when others =>" construct).

c. All loops for zero, one and many iterations, covering initialization, typical running and termination conditions.

Prior to the Test Readiness Review (TRR), a signed statement shall be provided verifying that items (1) through (3) have been satisfied. The test suite implementing (1) through (3) will be examined by the contracting agency during the TRR. Proposals shall describe, in Section 4.2.1 of the SDP, how structural test coverage will be implemented.

A.3 Regression Testing

Regression testing (identified as "retesting" in DoD-STD-2167A) is required for all operational software that undergoes change. Modifications of software to correct defects often result in the introduction of new defects. As high as 60% of code changes have been found to result in new defects [Humphrey 1989, p. 383]. Therefore, even the simplest, most trivial changes must be considered suspect, and the unit and systems that use it need to be retested. This contract clause extends the retesting required by DoD-STD-2167A to include structural coverage testing.

Suggested clauses for regression testing:

> Regression testing (described as "retesting" in DoD-STD-2167A) is required for all operational software that undergoes change. Regression testing shall include the functional tests required by DoD-STD-2167A and the complete set of structural coverage tests described in Item 2, above. All necessary regression testing shall be completed prior to system acceptance. Proposals shall describe, in Section 4.2.1 of the SDP, how regression testing will be implemented.

A.4 Testable Requirements

DoD-STD-2167A requires that all specified software requirements must be testable. This contract clause requires the developer to explain how they will ensure that this is the case. By focusing attention on this problem at an early stage, it attempts to avoid the introduction of meaningless requirements such as "shall be user friendly" and "shall give good performance" that cannot be tested.

Suggested clauses for testable requirements:

> DoD-STD-2167A requires that all software requirements documented in the System Requirements Specification (SRS) must be testable. A requirement is designed to determine whether the requirement is met by the software.

A.5 Automated Testing Tools

Performed manually, testing can be very labor intensive and is often incomplete. Automated testing tools can reduce the labor required and ensure that complete sets of applicable tests are run. There are many COTS tools available that support testing [Sittenauer 1991, Youngblut 1991]. This contract clause requires the developer to identify the testing tools currently used and those planned to be used on this project.

Required use of are automated testing tools is consistent with DoD policy. DoD Instruction 5000.2 states that "automated tools . . . should be used."

Suggested clauses for automated testing tools:

> Proposals shall describe, in Section 4.2.1 of the SDP, the methods and techniques that will be implemented to ensure that software requirements are testable. Proposals shall list, in Section 6.2.2 of the SDP, the automated tools that support static, dynamic and coverage analysis, and regression testing. Proposals shall identify those tools that are currently in use and those that are to be developed or acquired.

A.6 Error Cause Analysis

Error cause analysis enables contractors to determine why defects occurred and modify their development and quality assurance processes so that similar defects will not recur. Unless error cause analysis is performed the same problem can continue to cause new defects [Humphrey 1989, p. 364; Myers 1988, p. 93].

The requirement for error cause analysis is consistent with DoD policy. DoD Instruction 5000.2 states: ". . . ensure that the contactor establishes a uniform software defect data collection and analysis capability to provide insights into reliability, quality, safety, cost, and schedule problems. The contractor should use management information to foster continuous improvements in the software development process, to increase first time yields, to reduce test problems, and to reduce occurrences of software problem reports."

Suggested clauses for error cause analysis:

> Proposals shall describe, in Section 3.10 of the SDP, how error cause analysis will be used to identify systematic sources of defects and to improve software development and testing processes.

A.7 Reliability Analysis

Current methods of software reliability analysis track software defects throughout the development cycle and into operation. From this data estimates of the number of remaining defects and the probability of operational failures are derived. Although these techniques are relatively immature, they provide the only available concrete estimates of system reliability. In addition, the historical record of defect discovery and removal is expected to remain an important factor as reliability analysis techniques are improved. Since this data cannot be reconstructed after the fact, it must be collected as part of the development process.

Suggested clauses for reliability analysis:

> Reliability analysis of all operational software is required. Proposals shall describe, in Section 4.2.1 of the SDP, how reliability levels will be assessed. A report of the reliability assessments for each CSCI shall be provided prior to system acceptance.

A.8 Test Effectiveness and Progress

The use of a limited set of product and process measures can provide a program office with better insight into testing activities. This insight can reveal potential schedule or cost slippages when there is still time to take appropriate action. Defect detection rate is one fundamental measure. Here the basic mapping of the number of defects found per KLOC against time provides an indirect indication of both the growth in software quality and test progress. It can be extended in several ways. For example, the defect detection rate can be contrasted with the correction rate to indicate if debugging is becoming an obstacle to test progress. When an estimate of the total defects to be found is also included, the variance between defects estimated and corrected provides an estimate of progress towards completion. Additional measures can be used to focus attention on known problem areas. It is widely agreed, for example, that changes can be several times more defect prone than new code [Humphrey 1989, pp. 383, 439]. Consequently, in a test effort where frequent changes are required to accommodate evolving requirements, a manager might request data on the relative defect rate versus the percentage of modifications.

Tracking the test progress starts with knowledge of what is to be done and the results expected. In other words, a test plan that specifies the needed testing is a prerequisite for objectively assessing how much testing remains to be performed. Depending on the information provided in the test plan, there are a variety of measures that can be used. When the test plan specifies a required level of structural coverage as a stopping criteria, for example, the number of modules that have successfully reached the required coverage level provides an indicator of early testing progress. When specific test cases are defined, the total tests planned, test ready to run, tests run, and tests run successfully can be contrasted.

Suggested clauses for test effectiveness and progress:

> Proposals shall describe the measures that will be instituted to monitor test progress and software quality growth. Information on the corresponding data collection and analysis procedures, and supporting tools, will also be provided. In addition, once software design activities commence, the contractor shall provide bi-annual reports on trends on test effectiveness using both the data discussed here and that gained in the course of error cause analysis. The purpose of these reports will be to promote the identification of good development and testing practices so that other contractors can be encouraged to employ these approaches as appropriate.

REFERENCES

[Ackerman 1989]	Ackerman, A. Frank, Lynne S. Buchwald, and Frank H. Lewski. 1989. Software Inspections: An Effective Verification Process. *IEEE Software* 6/3 (May): 31-38.
[Alberts 1976]	Alberts, D.S. 1976. The Economics of Software Quality Assurance. *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY: AFIPS Press.
[Boehm 1970]	Boehm, Barry W. 1970. *Some Information Processing Implications of Air Force Space Missions: 1970-1980.* Memorandum RM-6213-PR: Rand Corporation.
[Boehm 1981]	Boehm, Barry W. 1981. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall.
[BP-SDP 1991]	*Software Development Plan for the Pre-Engineering and Manufacturing Development Phase of the Brilliant Pebbles System.* 20 November 1991. CDRL Sequence No. A008.
[Brykczynski 1990]	Brykczynski, Bill R., Reginald N. Meeson, and Christine Youngblut. October 1990. *A Strategic Defense Initiative Organization Software Testing Initiative.* IDA Paper P-2494. IDA: Alexandria, VA.
[Cohen 1985]	Cohen, Danny, study chairman. 1985. Eastport Study Group, *Summer Study 1985.*
[DeMillo 1987]	DeMillo, R.A., W.M. McCracken, R.J. Martin, and J.F. Passafiume. 1987. *Software Testing and Evaluation.* Redwood City, CA: Benjamin/Cummings Publishing Co.
[Dunn 1984]	Dunn, R.H. 1984. *Software Defect Removal.* NY: McGraw-Hill.
[Fagan 1976]	Fagan, Michael E. Design and Code Inspections to Reduce Errors in Program Development. 1976. *IBM Systems Journal* 15/3: 182-211.

[Fagan 1986] Fagan, Michael E. Advances in Software Inspections. *IEEE Transactions on Software Engineering* 12/7 (July 1986): 744-751.

[Fletcher 1983] Fletcher, James C., study chairman. 1983. *Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles*. SDIO: Washington, DC.

[Humphrey 1989] Humphrey, W.S. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.

[IEEE 1987] IEEE Computer Society. *Draft Standard of: A Standard Classification for Software Errors, Faults, and Failures*. December 1987. IEEE Computer Society.

[Kolkhorst 1988] Kolkhorst, B.G., and A.J. Macina. 1988. Developing Error-Free Software. *IEEE Aerospace and Electronic Systems* (November): 25-31.

[Luckham 1984] Luckham, D.C. and F.W. von Henke. September 1984. An Overview of ANNA, A Specification Language for Ada. Stanford University. CSL-TR-84-265. Also published in *IEEE Software* 2/2 (March 1985): 9-24.

[Luckham 1990] Luckham, D.C. 1990. *Programming with Specifications*. New York: Springer-Verlag.

[MDA 1991] *National Defense Authorization Act for Fiscal Years 1992 and 1993. Conference Report to Accompany H.R. 2100*. November 13, 1991. Washington D.C.: Government Printing Office.

[MoD 1991] United Kingdom. Ministry of Defense (MoD). *The Procurement of Safety Critical Software in Defence Equipment. Part 1: Requirements*. Ministry of Defence, Interim Defence Standard 00-55. 5 April 1991.

[Myers 1979] Myers, Glenford J. 1979. *The Art of Software Testing*. New York: John Wiley & Sons.

[Myers 1988] Myers, W. 1988. Shuttle Code Achieves Very Low Error Rate. *IEEE Software* (September): 93-95.

[Parnas 1985] Parnas, David L. 1985. Software Aspects of Strategic Defense Systems. *American Scientist* (September-October): 432-440.

[Russell 1991]	Russell, Glen W. 1991. Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software* 8/1 (January): 25-31.
[Sittenauer 1991]	Sittenauer, Chris, Greg Daich, Dolly Samson, Debbie Dyer, Gordon Price, John Hugie, and Gary Petersen. *Software Test Tool Report.* Hill AFB, UT: Software Technology Support Center.
[Weinberg 1984]	Weinberg, Gerald M. and Daniel P. Freedman. 1984. Reviews, Walkthroughs, and Inspections. *IEEE Transactions on Software Engineering* 12/1 (January): 68-72.
[Youngblut 1989]	Youngblut, Christine, Bill R. Brykczynski, John Salasin, Karen D. Gordon, and Reginald N. Meeson. February 1989. *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks.* IDA Paper P-2132. IDA: Alexandria, VA.
[Youngblut 1991]	Youngblut, Christine, Reginald N. Meeson, and Bill R. Brykczynski. October 1991. *An Examination of Selected Commercial Software Testing Tools.* Alexandria, VA: IDA Paper P-2628.

ACRONYMS

BE	Brilliant Eyes
BP	Brilliant Pebbles
CDR	Critical Design Review
COTS	Commercial off the Shelf
CRWG	Computer Resources Working Group
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DARPA	Defense Advanced Research Projects Agency
DID	Data Item Description
DOD	Department of Defense
DR	Discrepancy Report
FQT	Formal Qualification Testing
FSD	Federal Systems Division
GBI	Ground Based Interceptor
GBR	Ground Based Radar
GMD	Global Missile Defense
GPALS	Global Protection Against Limited Strikes
IDA	Institute for Defense Analyses
IEEE	Institute for Electrical and Electronics Engineers
KSLOC	K (thousand) Source Lines of Code
LCSAJ	Linear Code Sequence and Jump
MTTF	Mean Time to Failure
NASA	National Aeronautics and Space Administration
NMD	National Missile Defense
NRL	Naval Research Laboratories
NSF	National Science Foundation
NTDS	Naval Tactical Data System
NTF	National Test Facility
ONR	Office of Naval Research
RFP	Request For Proposal
SAGE	Semiautomated Ground Environment

SDC	Strategic Defense Command
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDS	Strategic Defense System
SDP	Software Development Plan
SEE	Software Engineering Environment
SOW	Statement of Work
SRS	System Requirements Specification
STD	Software Testing Description
STP	Software Test Plan
STR	Software Test Report
TMD	Theater Missile Defense
TRR	Test Readiness Review
US	United States